RCLAda, or Bringing Ada to the Robot Operating System

Alejandro R. Mosteo
Centro Universitario de la Defensa, Zaragoza, Spain.
Instituto de Investigación en Ingeniería de Aragón, Zaragoza, Spain; email: amosteo@unizar.es

Abstract

The Robot Operating System (ROS) is a commonly used framework in many fields of robotics research, with increasing presence in the industry. The next iteration of this framework, ROS2, aims to improve observed shortcomings of its predecessor like deterministic memory allocation and real-time characteristics. The officially supported languages in ROS2 are C++ and Python, although several other contributed APIs for other languages exist. RCLAda is an API and accompanying tools for the ROS2 framework that enable the programming of ROS2 nodes in pure Ada with seamless integration into the ROS2 workflow.

Keywords: Robotics Software Frameworks, Open Source, Ada 2012.

1 Introduction

The ROS project [1] was born after the experiences with the Player/Stage [2] robotic programming and simulation tools, lead in development by Willow Garage, with the objective of easing the programming of service robots and fostering a common platform for open source robotics. Today, many research institutions contribute and maintain components of ROS. The scope of ROS goes from build tools and communication libraries to reference-frame composition facilities among robotic parts. These core elements enable the bringing together of heterogeneous code bases, with which the community has built a myriad of almost plug-and-play components that provide hardware drivers and software algorithms. Most research-oriented robotic hardware comes with a ROS node that enables its integration out of the box.

ROS was not designed with safety and hard real-time considerations in mind, which has hindered its penetration in industrial domains, despite efforts like the ROSin [3] and ROS-industrial [4] projects. ROS2 design [5] squarely addresses these concerns, documenting dynamic memory allocation and threading characteristics of its API calls, and adopting the Data Distribution Service (DDS) standard [6] for data exchange.

There are three chief parts to ROS2 that give its flexibility and that a developer will be involved with. Firstly, there is a build system (Collective Construction [7], or *colcon* for short), that relies on XML metadata files to administer dependencies and orchestrate the parallel build of so-called packages. These packages, written in any programming language, constitute the second part and implement the functionality proper of ROS2. Their contents range from low-level hardware drivers for sensor and actuators to high-level algorithms that process and generate data, like planning and navigation strategies, visual recognition, SLAM [8], and so on. Finally, as the third part, there is a client library which is the API that enables seamless interaction between packages and the use of ROS2 concepts like topics, services and actions. This API is structured in a low level C API (RCL, from ROS2 Client Library), intended to enable the integration of other languages on an equal footing, but not necessarily as a direct target for programmers; and high-level client libraries that are bound to the RCL, of which the C++ and Python client libraries are the only officially supported by the ROS2 project, while several others in other languages (Java, C#, Objective C, Swift, Node.js) [9] are provided and supported by external contributors.

The RCLAda project targets the first and third parts just described; it is, fundamentally, an Ada binding to the RCL, supplemented by supporting integration facilities into the *colcon* toolchain. The RCLAda project has reached a level of completeness where it can already be used to write ROS2 packages using only Ada code, with some bits of CMake for the building integration.

This paper presents the design of the RCLAda project in the next section. Ada client packages and examples are shown in Sections 3 and 4, respectively. Facilities for integration into the build pipeline of ROS2 are presented next in Section 5, and conclusions in Section 6 close the paper.

2 Design

This section discusses several design aspects of both the Ada API and its integration in the build system.

2.1 Ada binding

At its core, RCLAda is a binding to a C library and, as such, has to deal with the pitfalls of such a binding: should it be a thin or a thick binding? Should it be partial or complete? Also, ROS2 is still an evolving project in which APIs are not frozen. How can be ensured that no breaking changes go unnoticed?

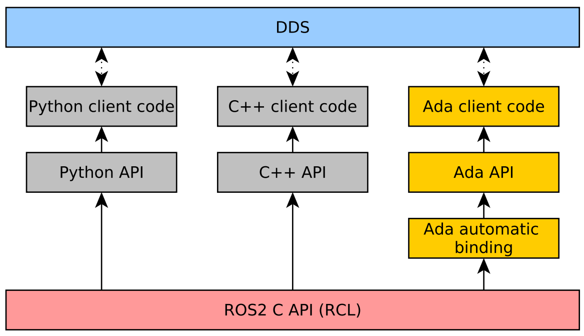

Figure 1: Dependencies and interactions among different kinds of code. At the bottom is the ROS2 Client Library, in C, intended as a common denominator on which all languages build its idiomatic API. The Ada API is a thick binding manually written on top of the autogenerated thin binding. ROS2 client code exchanges data through a common implementation of the Data Distribution Service, which is accessed through the RCL API too (i.e., there is no need to have a DDS API for every language).

In the past, there was little recourse in the Ada world but to manually create such a binding. This was how, for example, the precursor of ROS, the Player project, was made accessible to Ada [10]. Fortunately, the GCC Ada binding generator[1] has reached a point where, at least for the kind of C found in ROS2, its output is readily compilable. Preliminary tests with this generator suggested that the bulk of the mechanical work could be automated for the generation of a direct, low-level binding, with only minimal fixes needed in corner cases where GNAT would bug out even with proper source code generated by the binding generator.

This led to the adoption of a two-layer solution: a low-level thin binding is automatically generated that, consequently, is complete and a direct mapping to the RCL API, if not very Ada-developer friendly. On top of this binding, a thick, Ada-idiomatic one is provided. This solution eliminates the tedious manual work of generating a 1-1 mapping to a large number of C headers, and any changes in the underlying C API are immediately detected, as the thick Ada binding will fail to compile against the thin one.

This approach is reassuring in that the Ada code is safely bound to its underlying C counterpart by an automatic tool; and in that, in the worst case, a complete thin binding exists in case the thick binding were incomplete in some respect (no such need has been identified to date, though).

As it will be shown in the next subsection, the generation of the automatic binding is done as part of the build of the ROS2 code base. Each thin Ada binding is encapsulated within a GNAT project that supplies both the corresponding thick binding, when available, and any overriding files needed for the few cases in which a fix is needed for the automatic binding. To give an idea, in the version of RCLAda at the time of this writing, there are 60 C headers with a counterpart automatic Ada specification, of which only one[2] needed manual adjustments to circumvent a GNAT bug (for the version of the compiler in Ubuntu 18.04, which is the ROS2 supported development platform).

[1]Invoked with `gcc -fdump-ada-spec`.

[2]`https://github.com/ada-ros/rclada/blob/master/gpr_rcl/src/overrides/rcl_allocator_h.ads`

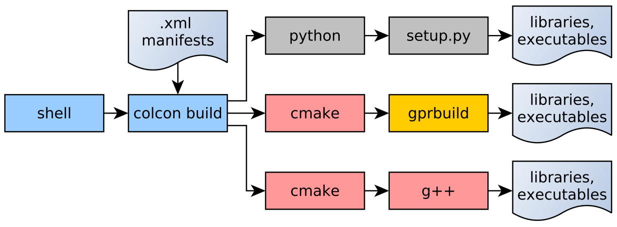

Figure 2: Simplified view of ROS2 build process with *colcon*. Packages may produce final executables or libraries to be linked in by other executables. For example, each package may define new message types, which are materialized as C data types and helper functions (a *typesupport* in ROS2 parlance) that are exported in one such library.

For the remainder of this paper there will be no more references to the low-level automatic bindings, since they are not used anywhere visible to clients of the thick Ada API. Fig. 1 schematizes the logical dependencies among code layers.

2.2 Allocators

ROS2 provides a way of tailoring dynamic memory allocation via *allocators*, which are passed as an extra argument to calls that allocate dynamic memory in the C API. Since Ada uses storage pools for this use case, a conscious effort has been done to relate both facilities. In practice, Ada storage pools can be transparently used as ROS2 allocators.

2.3 Build integration

ROS2 uses the *colcon* framework for building all its components. In order to be able to combine code coming from heterogeneous languages and organizations, each component is isolated in its own *package*. A package is thus a folder with an XML manifest which provides (simplifying) a list of dependencies on other packages, and a *colcon* build method. Dependencies are used to allow parallel compilation of independent packages, while the build method informs *colcon* on how the package is built (see Fig. 2). Readily available methods are:

- `CMake`: instructs *colcon* to look for a regular CMake `CMakeLists.txt` file to drive the build.
- `ament_CMake`: some extra ROS2-specific CMake macros are made available to the package that simplify finding build artifacts of other packages, and conversely exporting the package artifacts for dependent packages. This is particularly useful for the generation, importing and exporting of *messages*, which are the main data structures transmitted through DDS to exchange information among running ROS2 processes, as explained in Section 3.
- `python`: intended for Python packages, or packages that use Python's `Distutils setup.py`.

```
# LaserScan.msg:
# Single scan from a planar laser range-finder

std_msgs/Header header  # acquisition timestamp

float32 angle_min       # start angle of the scan [rad]
float32 angle_max       # end angle of the scan [rad]
float32 angle_increment # angular distance [rad]

float32 time_increment  # time between measurements [s]

float32 scan_time       # time between scans [s]

float32 range_min       # minimum range value [m]
float32 range_max       # maximum range value [m]

float32[] ranges        # range data [m]
float32[] intensities   # intensity data
                        # [device-specific units]
# If your device does not provide intensities,
# please leave the intensities array empty.
```

Figure 3: Example of a ROS2 message type definition.

Although *colcon* allows defining new build methods, given the current inhability of GNAT's `gprbuild` to execute other commands, or process environment information, RCLAda resorts to using the `ament_cmake` method, providing on top of it a few CMake functions that simplify the integration of GNAT's building toolchain. Details are presented in Section 5.

3 Ada Client Packages

This section delves into the available Ada packages within RCLAda that allow the creation of ROS2 programs. To be precise, ROS2 processes are called *nodes*. A node is one or several threads (typically together in a single executable) that provide some functionality and that are reachable by other nodes via *topics*, *services* and *actions*:

- Topics are named ROS2 addresses (e.g., `/robot/laser/readings` to which nodes can publish, to broadcast information, or subscribe, to receive information published to the topic. Each topic has an associated data type. Topics are used for data flows that go only from producer to consumers (e.g., sensor readings).
- Services implement a request/response mechanism, either in blocking or callback fashion. Services are used typically for node configuration calls, sending of data with acknowledgements, or other such two-way communications. Both the request and response have a different associated data type.
- Actions are used to provide high-level support for a sequence of interactions via service calls. Actions are just being introduced to ROS2 and were unavailable at the time of RCLAda first release, and for that reason they are not yet available in the Ada API.

```
declare
   Support : ROSIDL.Typesupport.Message_Support :=
             ROSIDL.Typesupport.Get_Message_Support
                (Pkg_Name, Msg_Type);
   --  This call retrieves the typesupport for a
   --  message type defined in a given ROS2 package.

   Msg : ROSIDL.Dynamic.Message := Init (Support);
   --  The Init call allocates a message. Receiving
   --  a message also requires specifying its type.
begin
   Msg ("valid").As_Bool := True;
   Msg ("range_min").As_Float32 := 1.0;
   --  Predefined types are accessed as equivalent Ada types.

   Msg ("ranges").As_Array.Resize (200);
   --  Dynamic arrays can be resized. Static arrays are
   --  allocated during initialization and cannot be resized.

   Msg ("ranges").As_Array (42).As_Int8 := 0;
   --  Array indexing of an array field. Indexing aspects
   --  of Ada 2012 are used to allow a more compact syntax.

   Msg ("image").As_Matrix ((100, 50, 1)).As_Int8 := 0;
   --  A matrix is indexed by providing a tuple of the
   --  appropriate dimensions.
end;
```

Listing 3.1: Example of use of messages. The general structure is to access a field by its name and give its expected type, which returns a reference to the value. There are specific indexing/resizing subprograms available for arrays and matrices.

Knowing this information, there are two main features in the ROS2 API: access to data types to be exchanged, and management of a node and its communications. From another point of view, these correspond to the static data type definitions and the run time node lifetime. ROS2 separates these features into two packages, which naming has been preserved in the Ada API to simplify the use of ROS2 documentation:

On the one hand, the `rosidl`[3] package deals with *typesupports*, which are C structs and associated functions for initialization/destruction of these data types. ROS2 types are defined in textual files (see Fig. 3) starting from a few predefined types (integers, reals, strings) that can be also composed into arrays or records with named fields. Every file defines one ROS2 type that results in the generation of an associated C struct and its initialization/destruction functions, stored in a linkable library that is available during the building of dependent packages. Packages, in turn, can reuse these types to define and export further types.

On the other hand, the `rcl` package deals with the creation of nodes, topics, services, and actions, and its use by communicating the aforementioned corresponding types through them. Next section will present examples in this regard.

3.1 The ROSIDL Ada packages

RCLAda provides a few Ada packages under the `ROSIDL.*` hierarchy that enable the use of ROS2 types; that is, an Ada node uses this hierarchy to *import* typesupports (in the sense of allocating a message of a given type, and accessing its

[3]ROS2 Interface Definition Language.

Package	Functionality
RCL.Allocators	ROS2 memory management.
RCL.Calendar	ROS2 clock types.
RCL.Clients	Client side of services.
RCL.Errors	ROS2 error codes.
RCL.Executors	Thread pools for nodes.
RCL.Logging	ROS2 logging system.
RCL.Nodes	Node registration.
RCL.Publishers	Publish side of topics.
RCL.Services	Server side of services.
RCL.Subscriptions	Subscribe side of topics.
RCL.Timers	ROS2 timers.

Table 1: Packages and functionality in the `RCL.*` hierarchy.

fields). Types are made available through subprograms in `ROSIDL.Typesupport`, while access to message fields is done through subprograms in `ROSIDL.Dynamic`.

The reason for the *dynamic* moniker is that there are two ways of using messages in ROS2. The dynamic way (currently used by RCLAda) is through an introspection library that provides a message in-memory layout at run time, and the static way is through generated C/C++/Python data types. RCLAda does not yet generate static Ada types to enable this usage.

Listing 3.1 shows how message fields can be accessed for reading and writing. With the exception of variable-length strings, that require the resizing of the underlying message, all fields are accessed through reference types, allowing minimal copying of information from/to messages. Also, bounds checking is performed as expected in an Ada context.

3.2 The RCL Ada packages

The rest, and bulk, of facilities to create a ROS2 node, and to interact with other nodes, are under the `RCL.*` hierarchy. Table 1 summarizes the names and functionality in each child package. The use of most of these packages will become clear with the examples presented in the next section. As an introduction, Listing 3.2 shows the creation of an empty node.

```
with RCL.Nodes;

procedure Null_Node is

   Node : RCL.Nodes.Node := RCL.Nodes.Init ("my_node");
   --  The Init call readies a node for use, and gives it
   --  the name with which it will be known to other nodes.

begin
   Node.Spin (During => Forever);
   --  Spin is a blocking call. If we had defined topics,
   --  services or other callbacks, these would be called
   --  as data became available. Also, information could
   --  be published to some topic from another thread.
   --  Alternatively, we could spin for a while and queue
   --  data whenever the node is not spinning.
end Null_Node;
```

Listing 3.2: A minimal null node.

4 Examples

ROS2 client libraries follow a tradition of implementing the same few basic examples. This allows users to compare idioms for the same functionality in different languages and these examples serve as a minimal tutorial. Two pairs of these examples, that work in tamdem, are presented in the next subsections. First, publish/subscribe nodes are shown (a topic), followed by a client/server interaction (a service).

4.1 Topics

The simplest form of communication in ROS2 are topics, in which one node publishes information that can be received by any number of other nodes. In the example collection, these nodes receive the names of *talker* and *listener*. The talker creates a topic and starts publishing a message every second. At any time, a listener can subscribe to said topic and it will start to receive the messages. Past messages are unknown to it.

Listings 4.1 and 4.2 show bare-bones code for these two classes of nodes. These nodes can obviously interact with their counterparts in other languages, e.g. the standard C++ and Python examples. A slightly more sophisticated version of the talker, using a ROS2 timer, can be found in the RCLAda repository.

4.2 Services

Services involve a request and a response, not necessarily of the same type. Both the server and the client must register a callback to be able to receive messages having an unknown arrival time. However, for the client, a blocking version exists that hides this callback, for simple use cases. The examples in Listings 4.3 and 4.4 exemplify this modus operandi for a service that receives two integers and returns their sum.

5 Integration

Up to this point, mostly Ada code has been presented. The compilation of such Ada code in the context of ROS2 requires the use of libraries (for ROS2 typesupports) which location cannot be known in advance; also, Ada nodes must be made known to the rest of the ROS2 ecosystem for some features to work seamlessly, like tab-completion, launching through ROS2 command-line tools, and other related issues. This section is devoted to present the RCLAda features that provide this integration.

Since the `ament_cmake` build mode of `colcon` was chosen, in practice this means that ROS2 packages containing Ada nodes must provide a `CMakeLists.txt` file that declares the libraries and executables being built, and the ROS2 data types being imported and exported. Since the GNAT compilation model is not at present supported by CMake, nor are the elaboration and binding stages well-suited to easy CMake integration, a number of CMake functions have been defined that forward the compilation to `gprbuild`. Also, given that importing messages requires the existence of GPR projects for their externally defined C libraries, these GPR projects can be generated on the fly too with approppriate CMake functions.

```
with RCL.Nodes;
with RCL.Publishers;

with ROSIDL.Dynamic;
with ROSIDL.Typesupport;

procedure Talker is
   Support : constant ROSIDL.Typesupport.Message_Support :=
               ROSIDL.Typesupport.Get_Message_Support
                 ("std_msgs", "String");
   --  Obtains the type for the messages to be published.
   --  This is the basic String type from ROS2.

   Node  : RCL.Nodes.Node := RCL.Nodes.Init ("talker");
   Pub   : RCL.Publishers.Publisher :=
             Node.Publish (Support, "/chatter");
   --  Creates a topic using the just initialized node.
   --  The topic requires a type and a hierarchical name.

   Count : Natural := 0; -- A counter of sent messages.
   Msg   : ROSIDL.Dynamic.Message :=
             ROSIDL.Dynamic.Init (Support);
begin
   loop
      Count := Count + 1;
      Msg ("data").Set_String ("Hello␣World:" & Count'Img);
      --  The String type contains a single field called
      --  data. Its value is updated here to reflect the
      --  amount of messages sent to date.

      Pub.Publish (Msg);
      --  Publish it to any subscribed listeners.

      Node.Spin (During => 1.0); -- Will drift...
      --  Spinning the node ensures events are dispatched.
      --  We take advantage of the spin period to wait 1s.
   end loop;
end Talker;
```

Listing 4.1: Talker example node.

```
--  withs omitted for conciseness.

procedure Listener is

   procedure Callback
     (Node : in out RCL.Nodes.Node'Class;
      Msg  : in out ROSIDL.Dynamic.Message;
      Info :        ROSIDL.Message_Info) is
   begin
      RCL.Logging.Info
        ("Got␣chatter:␣'" & Msg ("data").Get_String & "'");
      --  Print using the standard ROS2 logging facilities.
   end Callback;

   Node : RCL.Nodes.Node := RCL.Nodes.Init ("listener");

begin
   Node.Subscribe
     (ROSIDL.Typesupport.Get_Message_Support
                        ("std_msgs", "String"),
      "/chatter",
      Callback'Access);
   --  Subscribe to a topic. Whenever a message is received,
   --  the provided callback will be called with the data.
   --  The data type of publisher and subscriber must match.

   Node.Spin (During => Forever);
end Listener;
```

Listing 4.2: Listener example node.

```
--  withs omitted for conciseness.

procedure Server is
   -- Node declaration omitted for conciseness.

   procedure Adder
     (Node : in out RCL.Nodes.Node'Class;
      Req  :        ROSIDL.Dynamic.Message;
      Resp : in out ROSIDL.Dynamic.Message)
   --  The profile of a service includes the request
   --  made by the client and an initialized response
   --  of the appropriate type.
   is
      A : constant ROSIDL.Int64 := Req ("a").As_Int64;
      B : constant ROSIDL.Int64 := Req ("b").As_Int64;
   begin
      Resp ("sum").As_Int64 := A + B;

      --  Once the callback ends, the response will be
      --  sent to the requester automatically.
   end Adder;

begin
   Node.Serve
     (ROSIDL.Typesupport.Get_Service_Support
        ("example_interfaces", "AddTwoInts"),
      "add_two_ints",
      Adder'Access);
   --  Creating a service is as simple as giving the
   --  callback and the typesupport for the service,
   --  which in turn specifies the types of both the
   --  request and the response.

   Node.Spin (During => Forever);
end Server;
```

Listing 4.3: Server example node.

```
--  withs omitted for conciseness.

procedure Client is -- Synchronous version
   --  Node declaration omitted.

   Support : constant ROSIDL.Typesupport.Service_Support :=
               ROSIDL.Typesupport.Get_Service_Support
                 ("example_interfaces", "AddTwoInts");
   --  Note the specific Service_Support vs Message_Support.
   Request : ROSIDL.Dynamic.Message :=
               ROSIDL.Dynamic.Init (Support.Request_Support);
   --  The message is initialized using the Request_Support.
   --  An analogous Response_Support function exists.
begin
   Request ("a").As_Int64 := 2;
   Request ("b").As_Int64 := 3;
   --  Fill in the request data.

   declare
      Response : constant ROSIDL.Dynamic.Message :=
                   Node.Client_Call (Support,
                                     "add_two_ints",
                                     Request);
      --  By using the blocking call profile, the node is
      --  spun internally. An optional timeout can be used,
      --  and RCL_Timeout may be raised if expired.
   begin
      RCL.Logging.Info
        ("Got␣answer:" & Response ("sum").As_Int64.Image);
   end;
end Client;
```

Listing 4.4: Client example node.

```
project(my_ada_ros2_project VERSION 0.1.0)
# Standard CMake declaration of a project.

find_package(rclada_common REQUIRED)
# Mandatory to import the Ada CMake functions.

ada_begin_package()
# This, and its companion ada_end_package(), are in charge
# of setting up and propagating the Ada environment for this
# and dependent packages.

find_package(rclada REQUIRED)
find_package(rosidl_generator_ada REQUIRED)
# Standard calls to find the two ROS2 packages that contain
# the RCLAda thick bindings. Mandatory to write Ada nodes.

ada_add_executables(
       my_ada_project        # CMake target name
       ${PROJECT_SOURCE_DIR} # folder containing a GPR file
       bin                   # relative path to binaries
       my_ada_main)          # target binaries
# Associates a CMake target with a standard GNAT project.
# At make build time, gprbuild will run with proper context.
# The output executables will be made known to ROS2 tools.
# ${PROJECT_SOURCE_DIR} is provided by CMAKE, and points
# to the folder in which CMakeLists.txt is located; that
# is, to the root of the ROS2 package.
# The path for generated binaries is relative to the prev-
# ious discussed folder. Finally, a list of executables
# generated by the single GPR project file must be given.

ada_end_package()
```

Listing 5.1: Example of configuration file for CMake.

CMake relies heavily on environment variables to provide information about the build environment, like the location of libraries, headers, and so on. The same mechanism is used by RCLAda to propagate Ada-specific information, although this is done transparently to the user. Thus, the user can directly use the RCLAda projects and other generated GPR projects, since their actual locations are passed to `gprbuild` when invoked.

An example of `CMakeLists.txt` file presenting most available CMake Ada-specific functions can be seen in Listing 5.1. In addition, an equivalent function to `ada_add_excutable` devoted to exporting libraries exists (named `ada_add_library`). Finally, used internally by RCLAda, but probably useful to people writing nodes that require bindings to C libraries (e.g., for hardware drivers), the `ada_generate_binding` enables the generation at compile-time of an automatic thin binding and its integration into a given GPR project.

6 Conclusions

ROS and ROS2 have positioned themselves as the default go-to frameworks for research in service robotics. ROS2 strives to fix the shortcomings identified in ROS, and is expected to continue playing a significant and growing role in robotics research and industries in the years to come. For this reason, making Ada readily available in its ecosystem is a pursuit that ought to benefit roboticists with interest in a safer, time-proven language such as is Ada, and promote the language in a popular open source context. ROS2 aims to be more amicable to real-time and safety-critical robotics deployments even outside of the research community, for which Ada and SPARK are ideal client languages.

RCLAda enables the immediate writing of ROS2 nodes in Ada, without the hassle of having to create custom bindings or mixed-language programs, nor of having to battle a build system in which the GNAT toolchain is not trivial to be integrated. To this end, RCLAda provides both an Ada thick-binding to the ROS2 API, and a set of CMake functions for the ROS2 build system. Examples for the use of basic features of both RCLAda and ROS2 are provided that should help newcomers in the writing of new Ada ROS2 nodes.

RCLAda is available under an open source license to interested parties at `https://github.com/ada-ros/ada4ros2/`.

Acknowledgements

This work has been supported in part by research projects RoboChallenge (DPI2016-76676-R-AEI/FEDER-UE), Asimov (CUDZ2018-03), RoPeRT (DGA-T45_17R/FSE), and the AdaCore company.

References

[1] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng (2009), *ROS: an open-source Robot Operating System*, in ICRA workshop on open source software, vol. 3, p. 5, Kobe, Japan.

[2] B. Gerkey, R. T. Vaughan, and A. Howard (2003), *The Player/Stage project: Tools for multi-robot and distributed sensor systems*, in Proceedings of the 11th Int. Conf. on Advanced Robotics, vol. 1, pp. 317–323.

[3] Fraunhofer IPA, *ROSin: ROS-industrial quality-assured robot software components*, https://rosin-project.eu/. Accessed: 2019-Sep-06.

[4] S. Edwards and C. Lewis (2012), *ROS-industrial: applying the robot operating system (ROS) to industrial applications*, in IEEE Int. Conference on Robotics and Automation, ECHORD Workshop.

[5] Open Source Robotics Foundation, Inc, *ROS2 design*, http://design.ros2.org/. Accessed: 2019-Sep-06.

[6] G. Pardo-Castellote (2003), *OMG data-distribution service: Architectural overview*, in Proceedings of the 23rd International Conference on Distributed Computing Systems Workshops, pp. 200–206, IEEE.

[7] D. Thomas, *Collective construction (colcon)*, https://github.com/colcon. Accessed: 2019-Sep-06.

[8] ROS Index, *Available packages for ROS2 Dashing*, https://index.ros.org/packages/#dashing. Accessed: 2019-Sep-06.

[9] ROS Index, *About ROS2 client libraries*, https://index.ros.org//doc/ros2/Concepts/ROS-2-Client-Libraries/. Accessed: 2019-Sep-06.

[10] A. R. Mosteo and L. Montano (2007), *SANCTA: An Ada 2005 general-purpose architecture for mobile robotics research*, in International Conference on Reliable Software Technologies, pp. 221–234, Springer.